Python REGULAR EXPRESSIONS

Session - 5

What is mean by Regular Expression?

Regular expressions are a powerful language for matching text patterns and standardized way of searching, replacing, and parsing text with complex patterns of characters

All modern languages have similar library packages for regular expressions i.e., ***re*** built in module

Features of Regex

- Hundreds of code could be reduced to a one line elegant regular expression
- Used to construct compilers, interpreters and text editors
- Used to search and match text patterns
- Used to validate text data formats especially input data
- Popular programming languages have Regex capabilities

  Python, Perl, JavaScript, Ruby ,Tcl, C++,C#

General uses of regular expressions to:

- Search a string (search and match)
- Replace parts of a string (sub)
- Break string into small pieces (split)
- Finding a string (findall)

Before using the regular expressions in your program, you must import the library using "import re"

General Concepts

- Alternative : |
- Grouping : ()
- Quantification : ?*+{m,n}
- Anchors : ^ $
- Meta- characters : . [][-][^]
- Character classes : \d\D\w\W…..

Alternative:

Eg: “cat|mat” ==“cat” or “mat”

“python|jython” ==“python” or “jython”

Grouping:

Eg: gr(r|a)y==“grey” or “gray”

“ra(mil|n(ny|el))”==“ramil” or “ranny” or “ranel”

Quantification

? == zero or one of the preceding element

Eg: “rani?el”==“raniel” or “ranel”

“colou?r”==“colour” or “color”

* == zero or more of the preceding element

Eg: “fo*ot”==“foot” or “fooot” or “fooooooot”

“94*9” ==“99” or “9449” or “9444449”

+ == one or more of the preceding element

Eg: “too+fan”==“toofan” or “toooooofan”

“36+40”==“3640” or “3666640”

{m,n} == m to n times of the preceding element

Eg: “go{2,3}gle”== “google” or “gooogle”

“6{3} “==“666”

“s{2,}”==“ss” or “sss” or “ssss” ……

Anchors

^ == matches the starting position with in the string

Eg: "^obje"=="object" or "object – oriented"

"^2014"=="2014" or "2014/20/07"

$ == matches the ending position with in the string

Eg: "gram$"=="program" or "kilogram"

"2014$" == "20/07/2014", "2013-2014"

Meta-characters

.(dot)== matches any single character

Eg: "bat."== "bat" or "bats" or "bata"

"87.1"=="8741" or "8751" or "8761"

[]== matches a single character that is contained with in the brackets

Eg: "[xyz]" == "x" or "y" or "z"

"[aeiou]"==any vowel

"[0123456789]"==any digit

[-] == matches a single character that is contained within the brackets and the specified range.

Eg: "[a-c]" == "a" or "b" or "c"

"[a-zA-Z]" == all letters (lowercase & uppercase)

"[0-9]" == all digits

[^] == matches a single character that is not contained within the brackets.

Eg: "[^aeiou]" == any non-vowel

"[^0-9]" == any non-digit

"[^xyz]" == any character, but not "x", "y", or "z"

Character Classes

Character classes specifies a group of characters to match in a string

\d ⟶ Matches a decimal digit [0-9]

\D ⟶ Matches non digits

\s ⟶ Matches a single white space character **[\t-tab,\n-newline**, \r-return,\v-space, \f-form]

\S ⟶ Matches any non-white space character

\w ⟶ Matches alphanumeric character class ([a-zA-Z0-9_])

\W ⟶ Matches non-alphanumeric character class ([^a-zA-Z0-9_])

\w+ ⟶ Matches one or more words / characters

\b ⟶ Matches word boundaries when outside brackets.
Matches backspace when inside brackets

\B ⟶ Matches nonword boundaries

\A ⟶ Matches beginning of string

\z ⟶ Matches end of string

The search function

Search scans through the input string and tries to match at any location

The search function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

syntax :

```
re.search(pattern, string, flags=0)
```

Pattern --This is the regular expression to be matched

String-- This is the string, which would be searched to match the pattern anywhere in the string

Flags-- You can specify different flags using bitwise OR (|). These are modifiers.

Modifier	Description
re.I	Performs case-insensitive matching
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.u	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.x	It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

OPTION FLAGS

The *re.search* function returns a **match** object on success, **None** on failure. We would use *group(num)*or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple

- *match object* for information about the matching string. *match object* instances also have several methods and attributes; the most important ones are

Method	Purpose
group( )	Return the string matched by the RE
start( )	Return the starting position of the match
end ( )	Return the ending position of the match
span ( )	Return a tuple containing the (start, end) positions of the match

Sample Program - 1 for search function

```
""" This program illustrates is
    one of the regular expression method i.e., search()"""
import re
# importing Regular Expression built-in module
text = 'This is my First Regulr Expression Program'
patterns = [ 'first', 'that‘, 'program' ]
# In the text input which patterns you have to search

for pattern in patterns:
    print 'Looking for "%s" in "%s" ->' % (pattern, text),
    if re.search(pattern, text,re.I):
        print 'found a match!'
     # if given pattern found in the text then execute the 'if' condition
    else:
        print 'no match'
     # if pattern not found in the text then execute the 'else' condition
```

Modifying Strings

- Till now we have done simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways using the following pattern methods.

Method	Purpose
Split ( )	Split the string into a list, splitting it wherever the RE matches
Sub ( )	Find all substrings where the RE matches, and replace them with a different string
Subn ( )	Does the same thing as sub(), but returns the new string and the number of replacements

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions

Split Method:

Syntax:

re.split(string,[maxsplit=0])

Eg:

```
>>>p = re.compile(r'\W+')
>>> p.split(‘This is my first split example string')
[‘This', 'is', 'my', 'first', 'split', 'example']

>>> p.split(‘This is my first split example string', 3)
[‘This', 'is', 'my', 'first split example']
```

Sub Method:

The sub method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method would return modified string

Syntax:

re.sub(pattern, repl, string, max=0)

Sample Program for Sub Method

```
import re
DOB = "25-01-1991 # This is Date of Birth“
# Delete Python-style comments
Birth = re.sub (r'#.*$', "", DOB)
print "Date of Birth : ", Birth
# Remove anything other than digits
Birth1 = re.sub (r'\D', "", Birth)
print "Before substituting DOB : ", Birth1
# Substituting the '-' with '.(dot)'
New=re.sub (r'\W',".",Birth)
print "After substituting DOB: ", New
```

Match Methods

- There are two types of Match Methods in RE
  1) Greedy
  2) Lazy
- Most flavors of regular expressions are greedy by default
- To make a Quantifier Lazy, append a question mark to it
- The basic difference is ,Greedy mode tries to find the last possible match, lazy mode the first possible match.

Example for Greedy Match

```
>>>s = '<html><head><title>Title</title>'
>>> len(s)
 32
>>> print re.match('<.*>', s).span( )
(0, 32)   # it gives the start and end position of the match
>>> print re.match('<.*>', s).group( )
<html><head><title>Title</title>
```

Example for Lazy Match

```
>>> print re.match('<.*?>', s).group( )
<html>
```

Explanation for Examples

- The RE matches the '<' in <html>, and the .* consumes the rest of the string. There's still more left in the RE, though, and the > can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the '<' in <html>to the '>' in </title>, which isn't what you want.
- In the Lazy (non-greedy) qualifiers *?, +?, ??, or {m,n}?, which match as *little* text as possible. In the example, the '>' is tried immediately after the first '<' matches, and when it fails, the engine advances a character at a time, retrying the '>' at every step.

Findall:

- Return all non-overlapping matches of *pattern* in *string*, as a list of strings.
- The *string* is scanned left-to-right, and matches are returned in the order found.
- If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group.
- Empty matches are included in the result unless they touch the beginning of another match.

Syntax:

re.findall (*pattern, string, flags=0*)

The findall example

```
import re
line='Python Orientation course helps professionals fish best opportunities‘
Print “Find all words starts with p”
print re.findall(r"\bp[\w]*",line)

print "Find all five characthers long words"
print re.findall(r"\b\w{5}\b", line)

# Find all four, six characthers long words
print “find all 4, 6 char long words"
print re.findall(r"\b\w{4,6}\b", line)

# Find all words which are at least 13 characters long
print “Find all words with 13 char"
print re.findall(r"\b\w{13,}\b", line)
```

Finditer

- Return an *iterator* yielding MatchObject instances over all non-overlapping matches for the RE *pattern* in *string*.
- The *string* is scanned left-to-right, and matches are returned in the order found.
- Empty matches are included in the result unless they touch the beginning of another match.

Syntax:

re.finditer(*pattern*, *string*, *flags=0*)

findall and finditer program

```
import re
string="Python java c++ perl shell ruby tcl c c#"
print re.findall(r"\bc[\W+]*",string,re.M|re.I)
print re.findall(r"\bp[\w]*",string,re.M|re.I)
print re.findall(r"\bs[\w]*",string,re.M|re.I)
print re.sub(r'\W+',"",string)
it = re.finditer(r"\bc[(\W\s)]*", string)
for match in it:
    print "'{g}' was found between the indices
     {s}".format(g=match.group(),s=match.span())
```